*Carl Kingsford, 02-201, Fall 2015*

# Lecture 16: Object-oriented Programming

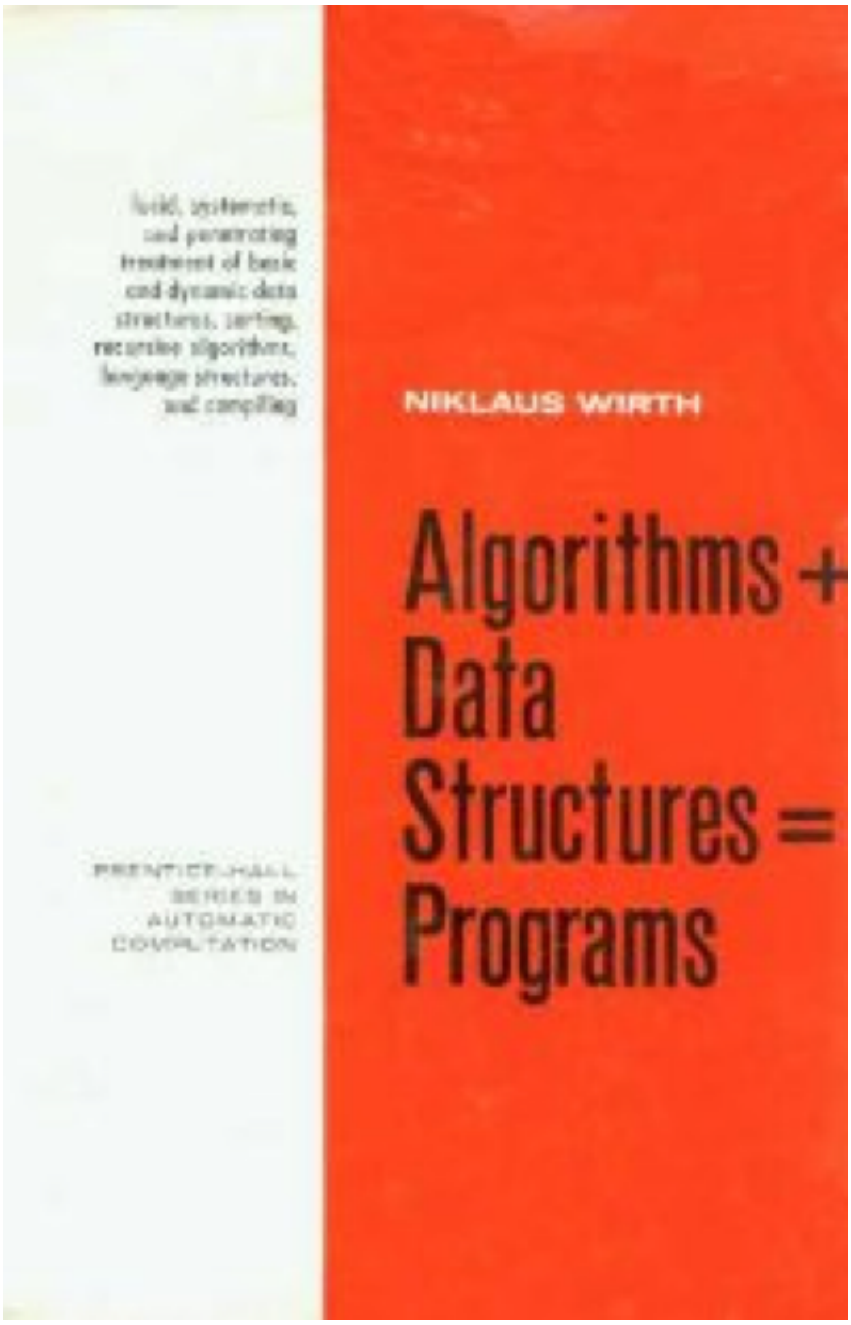
Programming languages provide many ways to help you organize your code. Functions are the most important and we've seen a lot of examples of them. Functions help you organize the "action" of your program --- functions are like new verbs that you create. This organization has helped us use "top-down" design: start from the big problem and break it into smaller problems, writing a function for each of the smaller problems.

We've also seen that it is often helpful to start your program by thinking about how you will organize your data, and we've seen data structures such as lists, maps, structs, and pointers that help you do this. This leads to another useful way of thinking about how to design your program: describe the organization of your data and have that reflected in your program. For example:

- A contact management program will manipulate Contacts
- A drawing program will manipulate a Canvas, and perhaps Lines, Colors, and Shapes
- Facebook will manipulate Users, Posts, and Advertisements
- Twitter will manipulate Tweets, Users, Advertisements

These are the "nouns" of these programs.

These two ways of thinking complement each other:




The techniques of object-oriented programming let you combine data organization with functions that operate on this data. It let's you think in terms of the "objects" that your program will manipulate (nouns) and the functions ("verbs") that perform actions using the data.

## Examples of noun/verb pairs

Once you've decided on the "nouns", you choose the "verbs" that apply to those nouns.

Here are some examples of pairs of "nouns" (data) with operations you might want to perform with them:

Your “noun” is a Tweet:

```
type Tweet struct {
    text string
    time uint64
    who *User
}
```

- Get Hashtags in Tweet
- Get Direct Mentions in Tweet
- Shorten URL in Tweet
- Get URLs in Tweet
- Get Short Version of Tweet

Your “noun” is a User:

```
type User struct {
    name string
    followers []*User
    following []*User
    tweets []*Tweets
}
```

- Direct Message User
- Add Follower
- Remove Follower
- Add Following User
- Remove Following User
- Get All Tweets from Followed Users

Your “noun” is a Contact:

```
type Contact struct {
    name string
    id int
    salary float64
    friends []*Contact
    phone []int
}
```

Operations you will need to perform on a Contact:

- Get First Name
- Get Last Name
- Set First Name
- Set Last Name
- Get Formatted Phone Number
- Call
- Count Friends
- Add Friend
- Give Raise

Here are some examples from the spatial games assignment:

```
type Field struct {
    cells [][]Cell
}
```

- Count Cell Kinds in Neighborhood
- Read Field From File
- Evolve Single Step
- Save Field To File
- Draw Field
- Check Field is Valid
- Zero All Scores

```
type Cell struct {
    kind  string
    score float64
    prevKind string
}
```

- Zero Score
- Set Kind
- Get Kind
- Get Previous Kind
- Get Cell Color

## The `Canvas` object

Go provides support for encoding the relationship between nouns and verbs in the form of "methods". You've already used methods when you used the `Canvas` object. Now we will see how this is implemented.

A `Canvas` is just a struct like you have already seen:

```
type Canvas struct {
    gc      *draw2d.ImageGraphicContext
    img     image.Image
    width   int
    height int
}
```

Pointer to an object that represents the pen

An object that represents the image

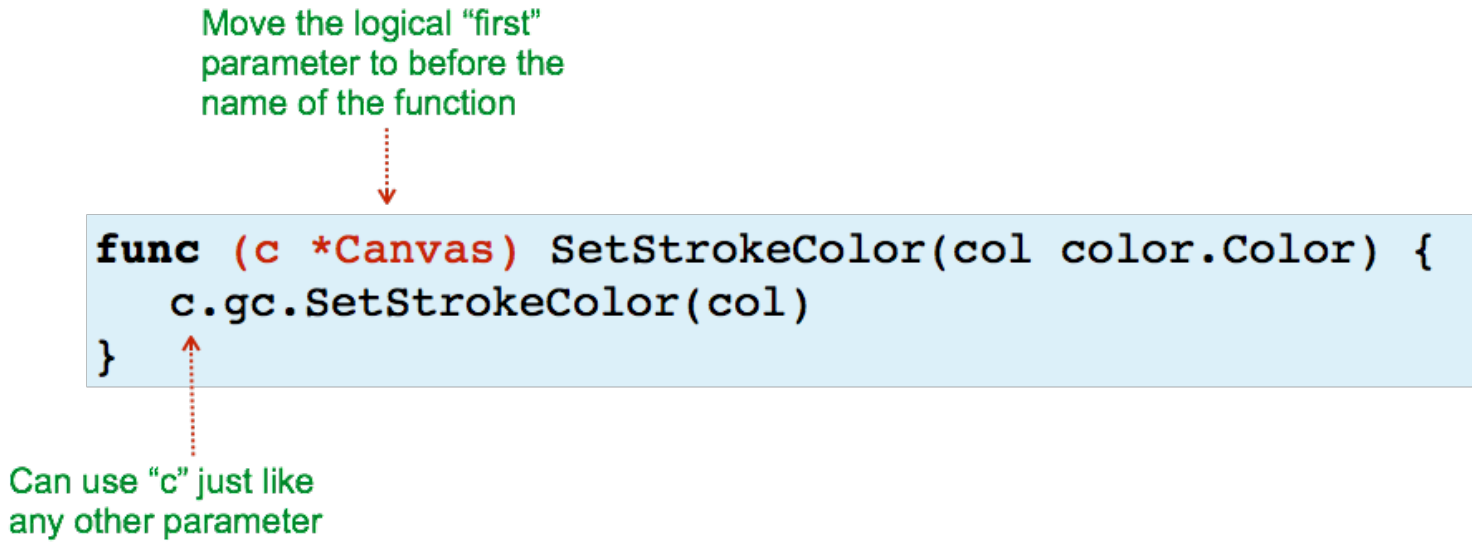
There are several operations that you want to perform on a `Canvas`:

```
    MoveTo(c *Canvas, x, y float64)
    LineTo(c *Canvas, x, y float64)
    SetStrokeColor(c *Canvas, col color.Color)
    SetFillColor(c *Canvas, col color.Color)
    SetLineWidth(c *Canvas, w float64)
    Stroke(c *Canvas)
    FillStroke(c *Canvas)
    Fill(c *Canvas)
    ClearRect(c *Canvas, x1, y1, x2, y2 int)
    SaveToPNG(c *Canvas, filename string)
    Width(c *Canvas)
    Height(c *Canvas)
```

These operations are logically related: they are the things you can do to a `Canvas`. As written above, they all take a `*Canvas` as their first parameter. Go provides a special syntax for this situation.

## Methods

A method is a function that operates on a particular type. Go's syntax for a method is the same as for a regular function with one addition:




Once you've defined a method, you call it using the "." syntax:

```
var pic *Canvas = MakeCanvas()
pic.SetStrokeColor(blue)
```

The parameter before the function name is called the receiver: it is the object that will "recieve" the effect of the

function.

So what's the point of methods? You can always do the same thing using a function, but the method syntax makes it clear that the function is a verb that operates on a particular noun. Itlogically groups operations with the data they operate on. It also supports the "noun" / "verb" way of designing programs directly.

This also let's you use the same function name for different object types. For example

```
(c *Canvas) Draw()
(b *Button) Draw()
```

are different functions: one draws a `Canvas` and one draws a `Button`.

We will also see that this way of organizing your code has some other useful benefits in re-using functions (next lecture).

> **Test yourself!** How could we re-write our translation simulator to use an object-oriented style?

# An example: Implementing a Stack

Recall our non-object-oriented implementation of a stack:

```
func createStack() []int {
   return make([]int, 0)
}

func push(S []int, item int) []int {
   return append(S, item)
}

func pop(S []int) ([]int, int) {
   if len(S) == 0 {
       panic("Can't pop empty stack!")
   }
   item := S[len(S)-1]
   S = S[0:len(S)-1]
   return S, item
}
```

This was used in the following way:

```go
func main() {
    S := createStack()

    S = push(S, 1)
    S = push(S, 10)
    S = push(S, 13)
    fmt.Println(S)

    S, item := pop(S)
    fmt.Println(item)

    S, item = pop(S)
    fmt.Println(item)

    S, item = pop(S)
    fmt.Println(item)
}
```

The above code is kind of ugly because we have to pass `S` to the function each time, and also make sure to reassign the result of the function calls to `S`. Using methods can make this code shorter and clearer.

**Step 1.** Define a type that corresponds to our noun that can hold the data we need for a stack:

```go
type Stack struct {
    items []int
}
```

**Step 2.** Define methods for the verbs: Push, Pop:

```go
func (S *Stack) Push(a int) {
    S.items = append(S.items, a)
}

func (S *Stack) Pop() int {
    a := S.items[len(S.items)-1]
    S.items = S.items[:len(S.items)-1]
    return a
}
```

**Step 3.** Define a regular function for Create:

```go
func CreateStack() Stack {
    return Stack{items: make([]int, 0)}
}
```

In order to call a method, we need a variable of the appropriate type. So `Create` can't be a method since that is how we create a variable of this type. This is sometimes called a *factory function* since it creates variables of a given type.

Now using the stack is much nicer:

```go
S := CreateStack()
S.Push(10)
S.Push(20)
fmt.Println(S.Pop())====
```

# Points on Pointers and Methods

Suppose you have a type:

```go
type Date struct {
    month, day, year int
}
```

You want to write a method called `NextDay()` that will set the date to the next day from what it currently is. You would write a function with the following signature:

```go
func (d *Date) NextDay() {
    d.day++
    switch d.month {
    case 1: if d.day == 32 {
        d.month = 2
        d.day = 1
    }
    // and do on
    }
}
```

Notice that we made the receiver have type `*Date`. This is for the normal reason: we want to *change* the

date, so we need a pointer to the date we want to change, rather than a copy.

Now suppose we want to call this function. The "logically correct" syntax would be:

```
var today Date = Date{day:4,month:11,year:2015}

(&today).NextDay()
```

`today` is of type `Date`, so `&today` is of type `*Date`, which is the correct type for the receiver. However `(&today).` is a lot to type, so Go allows you to abbreivate this as:

```
today.NextDay()
```

Go knows that the receiver should be a pointer to `today`, so it automatically does the `&today` conversion.

# Summary

- Methods are functions that are associated with a type.
- If you have a variable `X`, you can call any of its methods using:

```
X.methodName(param1, param2)
```

This works like a normal function call.

- Writing your code this way is object-oriented programming.
- The way to design a program in this style is to:
    - Create types for the things your program will manipulate
    - Write methods for each of those types that perform the operations on those things that you will need.
    - Use those methods to solve the tasks you are aiming to solve.

# Glossary

- method: a function that operates on a particular tpe
- object-oriented programming: the style of programming that defines "objects" (nouns) and the verbs

(functions) that operate on them explictly.